Primórdios das Arquiteturas em Camadas

- Anos 90 – primeiros sistemas “em camadas”
  - Apenas 2 camadas: cliente – servidor
  - Cliente = interface com o usuário
  - Servidor = BD relacional
- Ferramentas para o desenvolvimento dos clientes: Visual Basic, PowerBuilder, Delphi
- Facilitavam a construção de aplicações de manipulação intensiva de dados
- Permitiam que controles fossem arrastados para uma área de desenho da interface e que depois fossem conectados a elementos do BD

Problema da Arquitetura Cliente-Servidor

- Onde embutir a lógica do domínio?
  - Regras de negócio, validações, cálculos, etc.
- Geralmente, ficavam no código do cliente
  - Lógica era embutida nas telas da interface
  - Precisava ser replicada em diferentes telas → manutenção difícil

Problema da Arquitetura Cliente-Servidor

- Alternativa: embutir lógica do negócio no servidor → no BD, por meio de *stored procedures*
- Essa estratégia não é muito bem aceita devido as características das linguagens para *stored procedures*
  - São mais pobres que linguagens de programação convencionais
  - Não são padronizadas; são específicas para um SGBD e impedem que o BD possa ser "portado" a um baixo custo

Solução: Arquitetura em 3-Camadas

- Contexto: disseminação das LPOOs e o crescimento da Web
- Componentes:
  - **Camada de apresentação**: para as interfaces com o usuário
  - **Camada de domínio**: para a lógica do negócio
  - **Camada de fonte de dados** (*data source*)

Arquiteturas para a Comunicação com as Fontes de Dados

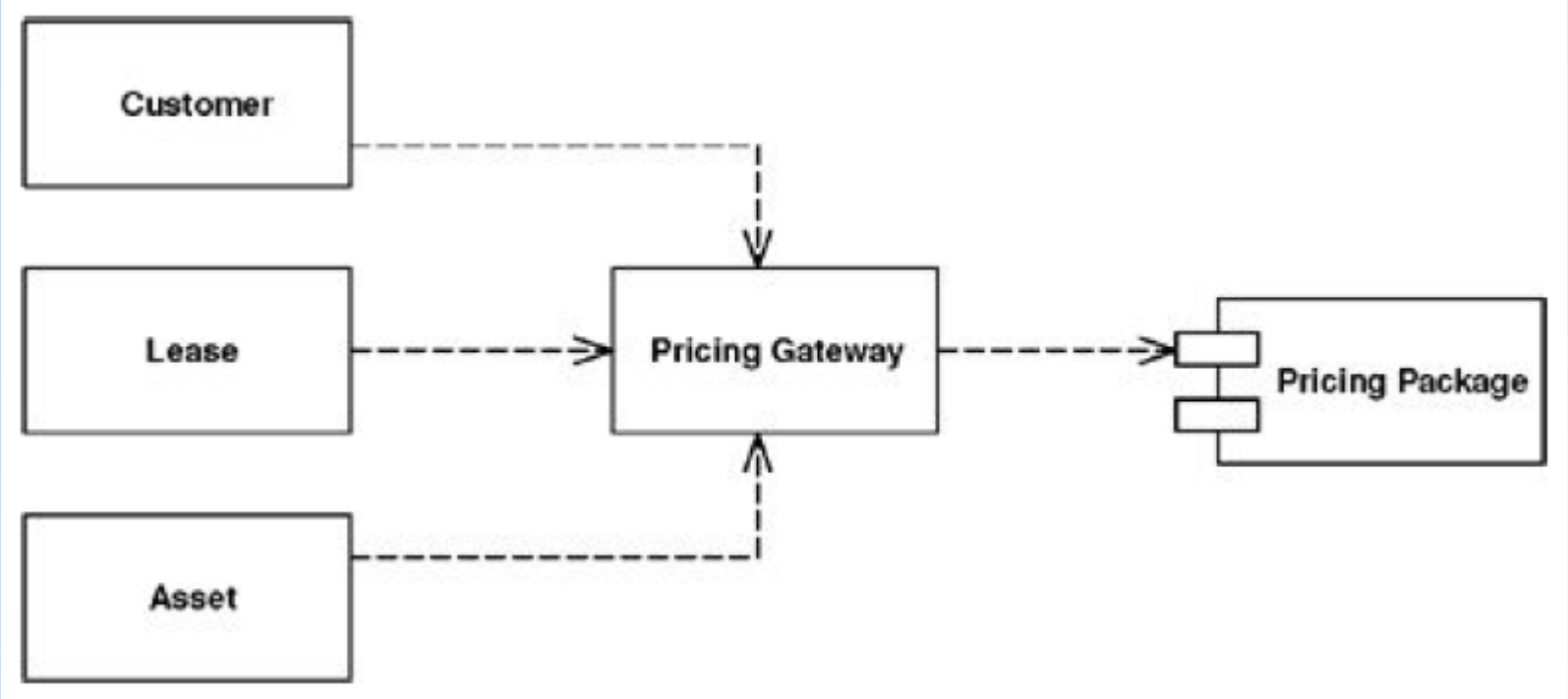
- Um dos principais papéis da camada de fonte de dados é estabelecer a comunicação entre o sistema e o BD
- Existem diferentes padrões arquiteturais no que se refere à forma como a lógica do domínio “conversa” com o BD
- A escolha de uma arquitetura para a comunicação com o BD é uma etapa crítica no projeto de um sistema, já que é algo muito complicado de se refatorar depois

Arquiteturas para a Comunicação com as Fontes de Dados

- Recomendado: separar código em SQL do código da lógica do negócio, colocando-os em classes diferentes
  - Vantagem dessa separação: código SQL concentrado em um só lugar evita replicação, facilita a manutenção, etc.
- Organização das classes pode se basear na estrutura do BD
  - Ex.: uma classe por tabela → *gateway* para a tabela

Gateway

- É um objeto que encapsula o acesso a um recurso ou sistema externo

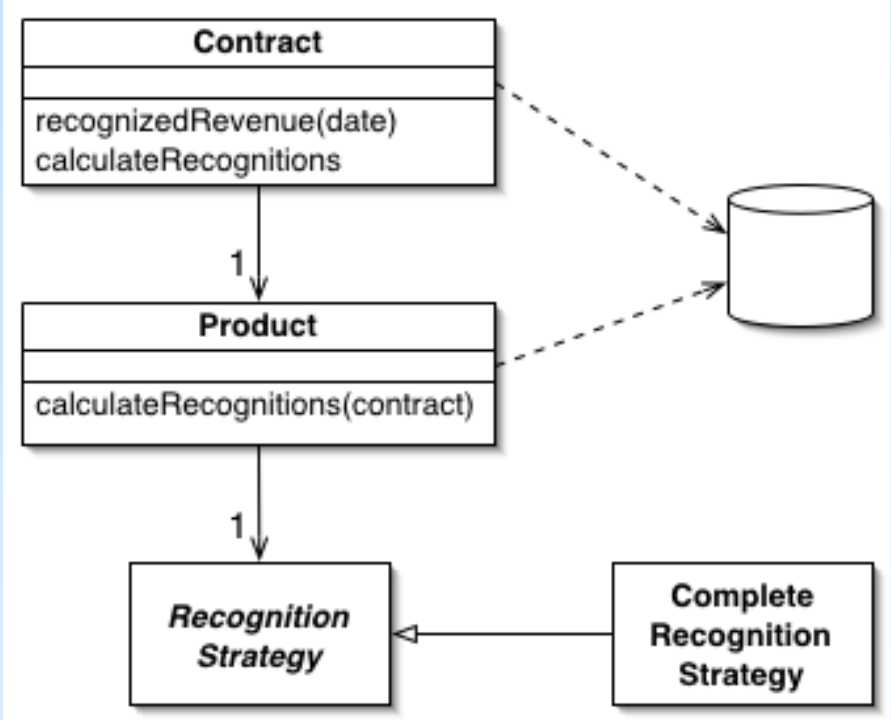


Objetos de Negócio

- Exemplos de objetos que abstraem conceitos de negócio:
  - Num sistema de processamento de pedidos:
    - Cliente, Pedido e Produto
  - Numa aplicação financeira
    - Cliente, Conta, Crédito, Débito
- Esses objetos modelam o domínio do negócio onde a aplicação específica irá operar → são chamados de **Modelo de Domínio**

Padrão Modelo de Domínio

- Um objeto do modelo de domínio incorpora tanto dados quanto comportamento
- Um modelo de domínio cria uma rede de objetos interconectados, onde cada objeto representa algum indivíduo significativo
  - Que pode ser tão grande quanto uma corporação ou tão pequeno quanto uma linha em um formulário de pedido



Persistência de Objetos

- Os objetos do modelo de domínio representam os principais estados e comportamentos da aplicação
- Geralmente, esses objetos:
  - São compartilhados por vários usuários simultaneamente
  - São armazenados e recuperados entre as execuções da aplicação
- Persistência de objetos – capacidade desses objetos de "sobreviverem" além do tempo de execução da aplicação
- Objetos podem ser persistidos em diferentes tipos de fontes de dados (*data sources*)
- **Obs.:** a persistência não é exclusividade dos objetos de domínio (mas é um requisito mais frequente para eles)

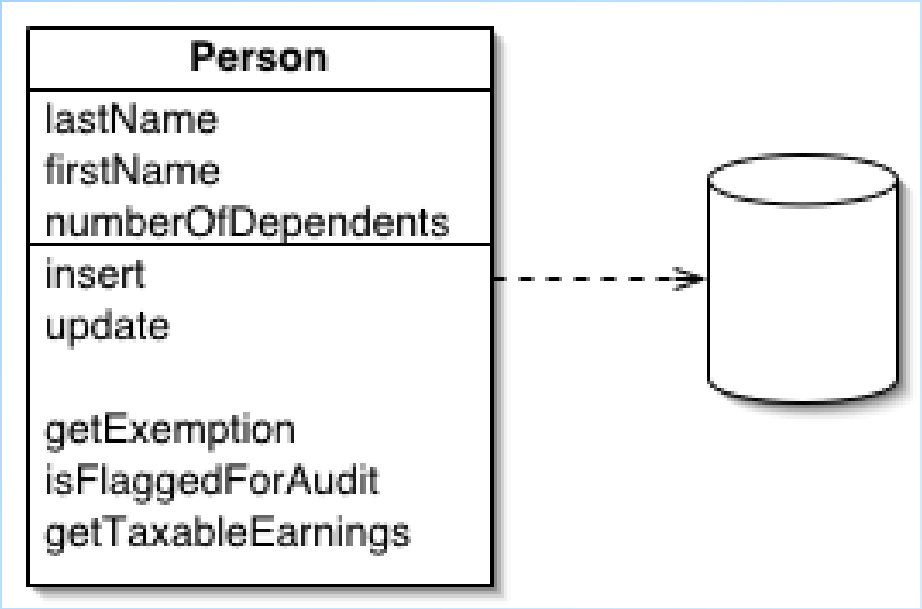
Padrões de Mapeamento para Modelo de Domínio

- Em aplicações que usam modelo de domínio, outras opções de mapeamento podem ser mais apropriadas
  - Dependendo do modelo de domínio e da estrutura da fonte de dados, os *gateways* podem ter complexidade **demais** ou **de menos**
- Exemplos de padrões de mapeamento usados com modelo de domínio:
  - ***Active Record***
  - ***Data Mapper***

Active Record

- Em aplicações com modelo de domínio simples:
  - Estrutura do modelo de domínio se assemelha bastante à do BD, com uma classe do domínio por tabela do BD
  - Objetos do domínio possuem lógica de negócio de complexidade moderada
  - Nesse contexto, é viável que cada objeto do domínio se ocupe pelo carregamento e pelo salvamento de dados no BD → esse padrão de arquitetura é chamado de **Registro Ativo (Active Record)**

Active Record



- Um objeto que encapsula uma tupla de uma relação do BD, encapsula o acesso ao BD e adiciona lógica de domínio aos dados
- Um objeto carrega tanto dados quanto comportamento
- A maior parte dos dados é persistente e precisa ser armazenada em um BD

O Padrão *Active Record* “na Prática”

- ActiveRecord (Ruby)
- PHP ActiveRecord (PHP)
- ActiveJDBC (Java)
- DBIC (Perl)
- Objective-C, Python, ...

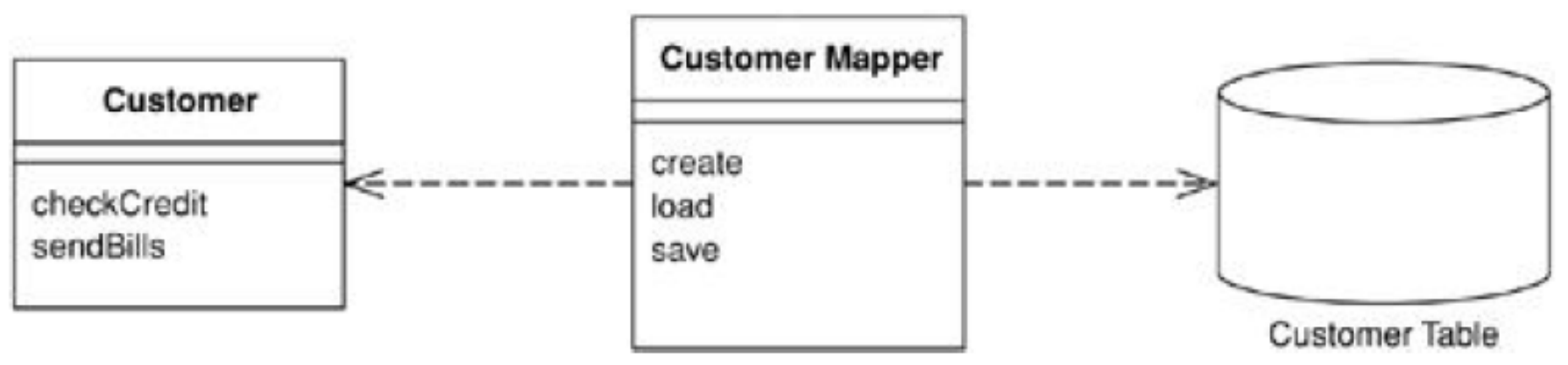
Mas o *Active Record* Não É Suficiente Quando...

- A lógica de domínio é mais complicada → modelo de domínio complexo
- Não há um mapeamento 1-para-1 entre as classes do modelo e as tabelas do BD
- Existe a necessidade de se testar a lógica de negócio sem que o acesso ao BD seja feito o tempo todo

  → Nesses casos, nem mesmo a indireção do padrão Gateway é suficiente para lidar com a complexidade
- Alternativa: padrão ***Data Mapper***

Data Mapper

- É uma camada de mapeadores que transferem dados entre objetos de domínio e um BD, mantendo-os independentes entre si
- Essa é a arquitetura de mapeamento mais complicada, mas que garante isolamento entre as duas camadas
  - Tanto o modelo de domínio quanto o de BD podem variar sem que um afete o outro



O Padrão *Data Mapper* “na Prática”

- MyBatis, para Java
- SQLAlchemy, para Python
- Ruby Object Mapper (ROM), para Ruby
- Doctrine, para PHP
- ...

Mapeamento Objeto – Relacional (ORM, de *Object Relation Mapping*)

- A maioria dos projetos de desenvolvimento de software usa:
  - Uma linguagem OO (como, Java, C#, Ruby, etc.)
  - Um BD relacional para armazenar dados
- Problema: **incompatibilidade conceitual** (*impedance mismatch*) entre objetos e relações
- Solução que não emplacou: uso de **Bancos de Dados de Objetos** (antes chamados de BDs Orientados a Objetos)
  - Não há implementações de SGBDOOs usadas em grande escala na atualidade
  - Mas muitos SGBDs se auto-denominam Objeto-Relacionais: PostgreSQL, Oracle, ...
- Solução mais moderna: uso de **Bancos de Dados NoSQL**
  - **ODM – *Object-Document Mapping***